CARDS: Conjoint Adaptive Retroactive Database System
by
Ashwin Krishnamurthy
Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science and Engineering at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 2002
© Ashwin Krishnamurthy, MMII. All rights reserved.
The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by
James Orlin
Professor
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Students
DISCLAIMER OF QUALITY

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available. If you are dissatisfied with this product and find it unusable, please contact Document Services as soon as possible.

Thank you.

Pages 48, 50, 52, 55, and 58 contain cropped text on the right-side margins. Best copy available.
CARDS: Conjoint Adaptive Retroactive Database System

by

Ashwin Krishnamurthy

Submitted to the Department of Electrical Engineering and Computer Science on May 24, 2002, in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science and Engineering

Abstract

The Conjoint Adaptive Retroactive Database System (CARDS), an alternative approach to adaptive conjoint analysis, was designed to minimize the effects of respondents’ burden and internal inconsistencies. Respondents were asked to sort web-based conjoint cards which formed a fractional factorial design. We forced our respondents to sort the cards in such a way that the orderings are perfectly consistent with feasible sort orders, which were calculated beforehand and are stored in a database. The simplifying assumptions inherent in CARDS allowed us to reach a solid estimate of utility functions quickly and efficiently.

Thesis Supervisor: James Orlin
Title: Professor
Acknowledgments

Professors Jim Orlin for providing me with this opportunity. Jim Orlin and Ely Dahan for their patience, insight, and support. Rob Hardy for his timely help.
3 System Overview

3.1 A High Level View of CARDS ........................................ 31

3.2 User Interface .......................................................... 33

3.3 Generating the Database .............................................. 34

3.3.1 Data Structures ..................................................... 34

3.3.2 Performing the Calculations ..................................... 35

3.3.3 Scoring the Cards ................................................ 37

3.3.4 Building and Managing the Buffer ......................... 38

3.3.5 Sorting ............................................................ 40

3.3.6 Observations ....................................................... 41

3.4 Software Issues ....................................................... 41

4 Conclusion & Future Work ............................................. 43

A C++ Code ............................................................... 47
List of Figures

1-1 Attributes P, Q, and R each have three levels associated with them. The product profile represents one possible combination (out of 27) of these attributes[3]. .................................................. 17

2-1 Full Factorial Design. .................................................. 25

2-2 The twelve conjoint cards used in the PDA study. ................. 26

3-1 A High level view of CARDS. ....................................... 32

3-2 Once each unique order in the buffer is processed, the ordering (along with accompanying statistics) are written to the database. ........ 40
List of Tables

2.1 The six binary attributes in the PDA study. .......................... 24

3.1 An $k \times n$ array, where $k$ is the number of levels and $n$ is the number of attributes. In CARDS, $k$ took on the value of either 10 or 20. ... 35

3.2 An $c \times n$ array, where $c$ is the number of cards and $n$ is the number of attributes. This represents the encodings for the first five cards of the conjoint study that were introduced earlier in Chapter 2. ............. 36

3.3 The buffer to which all card orders and their corresponding attributes are written. Each line represents an ordering resulting from one of the $k^n$ possible 5-tuples. .................................................. 38

3.4 The buffer to which all card orders and their corresponding attributes are written. Each line represents an ordering resulting from one of the $k^n$ possible 5-tuples. ................................. 39
Chapter 1

Introduction

Whether it’s digital cameras, plane tickets, DVDs, or organic food, consumer products and services are anything but simple in today’s marketplace. With increased technological sophistication and greater competitive pressures than ever before, products are getting more complex by the day. Cellular phones, for example, come in a variety of sizes, shapes, colors, and feature-sets.

A number of factors are responsible for this. First and foremost, products are designed to meet consumer needs; and today’s consumers demand more choice. Granted choice can prove useful when appropriate, choice often comes at the expense of simplicity. A simple example of this can be found at the local ice cream parlor. In the “old” days, ice cream came in three (maybe four) flavors, with the choice being limited to the customary chocolate, vanilla, and strawberry. Today, the typical ice cream shop boasts over 30 flavors, many of which are combinations of other flavors! The same concept holds true in many of today’s electronic products, including such gadgets as computers, laptops and PDAs. The basic business approach to selling equipment in this space is to offer more features, better performance, and more quality (scalability,
user experience, etc) at a lower price than the competition’s product. The result of this business approach is that products are more complex in order to support the additional features. Clearly, when purchasing products of increasing complexity, customers are faced with a daunting task.

On the flip side, those who design and develop today’s products are also faced with a rather difficult task. Even with a slowdown in consumer spending, deregulation and the introduction of discount players are eating away at gross margins in a number of business sectors. That, coupled with the increased pressures of including as many features (in a way that is affordable to consumers), necessitates design tradeoffs. Deciding which features are more important than others is by no means an exact science. Nevertheless, the companies that enjoy the most success are those that have a keen ability awareness of the features that drive the purchase decision. From a product development standpoint, such knowledge is critical and had many uses.

- **Pricing.** Knowing the features that consumers place a premium value on helps in setting the optimal price. For instance, knowledge on which features customers consider a “must-have” might indicate how sensitive they are to changes in price.

- **Design.** Gaining a true understanding of consumer preferences allows developers to design products in order to maximize customer satisfaction. The products that are designed with a true understanding of what is important to the customer will likely enjoy more success, thereby enhancing sales.

- **Cost Savings.** Product developers release products; consumers react to them. Rather than finding out if a product will be successful after it is released, a clear idea of what consumers want lessens the risk of suffering a loss due to a
marginally successful product. With prior knowledge, developers can make the proper adjustments before releasing a product.

Clearly, a central theme in marketing involves focusing on the customer. As mentioned earlier, customers today demand more choice. Companies in turn have to widen their ranges, while at the same time ensuring a turnover in inventory. This requires a great deal of market intelligence. The essence of marketing involves obtaining that intelligence. With improvements in technology and the ever-changing nature of customer needs, maintaining an awareness of customer preferences is an active task. Naturally, the best way to understand consumer preferences is to keep in touch with the consumers themselves. A number of marketing techniques involve the interacting with customers in order to better understand them, among which include product positioning and segmentation, product forecasting, test marketing and, conjoint analysis. In this project, we delved into the latter.

1.1 Conjoint Analysis

Conjoint analysis, one of the most popular and versatile marketing research techniques, has played a major role in providing an understanding of why consumers behave the way they do. The term “conjoint analysis” was coined by Green and Srinivasan in 1978 to refer to a number of paradigms in psychology, economics, and marketing that are concerned with the quantitative description of consumer preferences or value-trade-offs[3]. The idea behind conjoint is to mathematically model the way consumers behave when choosing between various products. Such a model would ideally reveal a significant amount of information about the customer. In particular, marketers and product developers need to know what features customers value most
(and least), as well as their relative importance when compared to one another. This information is particularly valuable when a decision must be made on which features to exclude.

The true value of conjoint, however, lies in measuring the tradeoffs that customers make when choosing the products they buy. Developing methods to measure tradeoffs among customer needs and/or features is, arguably, one of the most studied problems in marketing research[2]. Truly understanding the tradeoffs that consumers make requires a simulation – and that is precisely what conjoint entails. In essence, conjoint analysis uses hypothetical choice simulations generated according to the principles underlying the design of statistical experiments to measure individuals’ preferences, examine consumer choice behavior and/or predict their choice in new situations[7].

1.1.1 Components of a Conjoint Study

There are three main components in a conjoint study; all are illustrated in Figure 1-1.

- **Attributes.** First, the key attributes of a product must be determined. It is certain combinations of these attributes that are supposed to drive the purchase decision.

- **Levels.** Next, we determine the levels of each attribute. In CARDS, we chose to denote levels of attributes in “dollars.” That way, the level is indicative of the value the customer places on that particular attribute.

- **Product Profiles.** Product profiles represent different combinations of attributes. In conjoint experiments, individuals express their preference for various exper-
imentally designed, hypothetical alternatives. The product profiles represent these alternatives.

![Diagram](image)

Figure 1-1: Attributes P, Q, and R each have three levels associated with them. The product profile represents one possible combination (out of 27) of these attributes[3].

As shown in Figure 1-1, attributes represent the key elements of the product. It is certain combinations of these attributes that form the variations within a particular product. For instance, Compaq makes computers that come in different sizes, shapes, processor speeds, and memory configurations. In terms of a conjoint study, each type of computer Compaq sells can be represented by a profile. The product profiles in CARDS are represented in the form of “conjoint cards,” each card representing a particular combination of attributes. It is important to note that not all possible profiles are used in the study. Rather, a fraction of them are used; choosing which product profiles to use will be discussed in Chapter 2.

1.1.2 Utility Functions

As mentioned in Section 1.1, conjoint analysis models consumer preferences via hypothetical simulations. In order to simulate the decision-making process for a customer,
a conjoint study typically involves choosing from a number of product profiles and ranking them in the order of preference. It is this ranking that allows us to mathematically model the way consumers behave. In conjoint, consumer preferences are represented in terms of a utility function.

In order to analyze consumer preferences, marketers often employ an economic approach known as utility theory. Such an approach assumes that consumers choose between alternatives as if they have a pre-existing system in place for assigning preference scores (utilities) to each choice. When confronted with a choice, consumers always choose the option with the higher utility. A utility function specifies the utility (or well being) of a consumer for all combinations of attributes.

A linear program (LP) provides an intuitive description of a utility function. When deciding which product to purchase, consumers usually develop a set of rules that allows them to decide systematically. For instance, when purchasing a car, a consumer might realize that she always prefers a red car over a blue car. She may also prefer manual transmission over automatic. Rules such as these, in an LP-based approach, are known as constraints. Constraints are mathematical expressions that combine variables to express limits on the possible solutions[1]. Using the present example, we have the following constraints.

1. Red Cars are preferred over Blue Cars


Thus, it follows that:

\[ U_{red} > U_{blue} \]

\[ U_{manual} > U_{automatic} \]
Naturally, any LP-based model is linear, thereby limiting the constraints and combinations of variables to linear functions. Each individual utility such as \( U_{red}, U_{blue}, U_{manual}, \) and \( U_{automatic} \) are known as “part-worths.” For the purposes of CARDS, we assume that utility functions are linear and the attributes are independent from one another. Thus, utility function for a particular product is the sum of the part-worths of all the attributes. Going back to Figure 1-1, the utility for the profile is the sum of the utilities of attributes P, Q, and R.

\[
U_{profile} = U_P + U_Q + U_R = P_2 + Q_3 + R_1
\]

Utility functions are extremely useful in a conjoint study because they allow marketers to quantify preferences. Not only do they reveal which attributes consumers prefer over others, they shed light on exactly how much one attribute or product is preferred over another. Using the linear assumption (as we do in CARDS), differences in utility represent differences (or rank orders) in preference among the product[2].

### 1.2 Limitations of Conjoint Studies

As shown, in Section 1.1, conjoint analysis is extremely useful, allowing markets to arrive at a utility function that describes consumers’ preference models. Nevertheless, there are disadvantages. First, from a high level, it is difficult to ensure that the way customers behave in a simulation mirrors their actions in reality. There are a number of reasons for this. First, simulations usually involve pictures or descriptions of products rather than products themselves. With products such as houses and automobiles, it is rather infeasible for respondents to have to look at each and every one for a marketing study. Thus, it might be reasonable to expect some deviation
from actual behavior on the part of respondents. Without being able to interact with, and perhaps even try a product, it may be difficult for customers to get a feel for what they like.

Studies like this require effort on the part of respondents, often times more effort than they are typically willing to put forth. In order to obtain a clear indication of what consumers value and how they would react in various purchase situations, a great deal of information is needed. Requiring respondents to answer too many questions could result in less accurate data. Consequently, there is a tradeoff involved; a delicate balance exists between accuracy of results and the burden placed on respondents.

Moreover, consumer preferences might not be consistent with any particular mathematical model. In CARDS, we assume that utility functions display the property of linearity; but it is entirely possible for consumers to display preferences non-linear in nature. Thus, defining exactly what is consistent is a major decision in any conjoint study. In Chapter 2, we will address how the design of CARDS seeks to mitigate some of these effects.

1.3 Current Research

A great deal of research has been conducted in the realm of conjoint analysis. As Section 1.2 points out, conjoint studies often place a great deal of burden on respondents. Most of the research in conjoint has been pointed at reducing that burden, while still maintaining accuracy. The research can be separated into three main streams[2].

- Hybrid
- Hierarchical Integration
CARDS falls under the latter. In hybrid conjoint analysis, a combination of market and individual level data is used to arrive at the utility function. One approach that is gaining popularity in market research is Hierarchical Bayes. Unlike more popular estimation methods, Hierarchical Bayes (HB) random effects models do not require that individual level design matrices be of full rank, which leads to the possibility of using fewer profiles per subject than currently used[6]. In hierarchical integration, the respondents’ task is split into multiple stages, where the first stage operates at a very high level. For instance, one such study might involve first ranking the attributes in general, then proceeding to rank product profiles.

Conjoint studies that are adaptive are unique in the sense that current questions depend on the answers to the questions that preceded them. In other words, the experiment has a certain knowledge-base through which questions are devised based on the answers respondents provide. The premise behind adaptive conjoint analysis is to ask the question that gains the maximum amount of information. If the answer to one question reveals that $i$ is preferred to $j$, an adaptive approach would not allow the second question to reveal similar information. Rather, subsequent questions would reveal something new. An example of an adaptive approach involves showing the respondent pairs of profiles with various attribute combinations, and choosing the new combinations based on the choices of the previous ones. In most cases, an elaborate algorithm is used to determine the “next question,” based on previous answers.

CARDS takes an unconventional approach to adaptive conjoint analysis, as the primary algorithm at work in our system is enforcing consistency. In designing the system, certain parameters for consistency were clearly defined, and all orderings deemed “consistent” by the system were stored in a database, prior to the question-
ing. The design decisions that were made, as well as our notion of consistency, are discussed in Chapter 2.
Chapter 2

Design

In Chapter 1, some limitations of conjoint analysis were discussed. Among them were the excessive burden placed on respondents and internal inconsistencies. CARDS was designed specifically with those limitations in mind. The underlying goal of this system was to reduce the respondent burden, eliminate inconsistency, and arrive at customers’ utility functions as quickly and accurately as possible. This chapter outlines the features of CARDS as well as the choices that were made in its design.

2.1 PDA Study

The product that was used when CARDS was implemented was the PDA. The PDA is a product that is gaining in popularity and one, one with which most people are familiar and one that contains a wide spectrum of features.

Table 2.1 summarizes the attributes and levels that were used in the study. There were six attributes in total, each having 2 levels.
2.2 Profile Design

In order to arrive at the constraints such as those illustrated in Section 1.1.2, a conjoint study needs to reveal which attributes are preferred and quantify those preferences accurately. The brute force method might involve presenting the user with all combinations of attributes in the hypothetical choice simulation. Suppose a product had three binary attributes (each attribute having two levels) associated with it. Showing all combinations of attributes would require $2^3 = 8$ conjoint cards, where each card displays a particular profile. This is known as a full-factorial design. A design in which every setting of every factor appears with every setting of every other factor is a full factorial design. This design can be advantageous because it allows marketers to determine the effect of every attribute and the effects they have on one another. In Figure 2-1, illustrates a full factorial design. Since all combinations of attributes are evaluated by the respondent, we can accurately correlate the effect one attribute has on another. In multiattributed products, attributes often have a joint effect. Examples of joint constraints are:

- Customer wants the red color if and only if the shape is square.
- Customer will never buy small unless it is rectangular in shape.
Full factorial designs are ideal for studies that involve a small number of attributes. However, as the number of attributes increases, the cost of a study can become rather infeasible. For a study of \( n \) attributes with \( k \) levels, a full factorial design requires \( k^n \) cards. The number of cards is exponential in the number of attributes. Clearly, a full factorial design can grow quickly out of favor.

Furthermore, there is a tradeoff between the level of detail at the respondent level and the accuracy of results. Conjoint studies with many cards place a great deal of burden on the user. When the number of attributes is large, ranking is not only cumbersome, it can be confusing. Full-profile conjoint analysis procedures typically impose too much of an information overload on the respondents. When faced with such tasks, respondents resort to simplifying tactics and the results are likely to be distortions of the true preference structure[9]. To employ a full factorial design in CARDS would require 64 profiles – clearly an information overload.

To ease the respondent burden, CARDS utilizes a fractional factorial design, in which only a fraction of the 64 cards specified by the full factorial design are used. Various simplifying tactics are used when choosing which cards to include in the experiment. Profiles which will reveal very little about the consumers' preferences are typically eliminated from the start. For instance, certain combinations of attributes might be
infeasible of product development. It might be too expensive, or even technically impossible. In addition, certain extreme combinations of attributes might capture a negligible portion of the market, thereby allowing for their removal.

Certain combinations of attributes could either be infeasible in terms of product development, or unreasonable from a customer standpoint. Such profiles can be eliminated from the start. Another simplifying tactic involves ensuring two properties: balance and orthogonality. All good fractional factorial designs display both properties. We will show in Section 2.2.1 that 12 cards were used in our experiment. A balanced experiment has each attribute appearing an equal number of times. In an orthogonal design, we use a special fractional factorial called an “orthogonal array.” The details of this are beyond the scope of this paper.

2.2.1 PDA Conjoint Cards

The fractional factorial design allowed us to use 12 cards instead of the 64 called for in a full factorial design.

<table>
<thead>
<tr>
<th>large</th>
<th>small cell</th>
<th>large cell</th>
<th>small</th>
<th>large cell</th>
<th>large</th>
</tr>
</thead>
<tbody>
<tr>
<td>ext. battery</td>
<td>wireless</td>
<td>handsfree</td>
<td>ext. battery</td>
<td>wireless</td>
<td>handsfree</td>
</tr>
<tr>
<td>$499</td>
<td>$499</td>
<td>$249</td>
<td>$249</td>
<td>$249</td>
<td>$249</td>
</tr>
</tbody>
</table>

Figure 2-2: The twelve conjoint cards used in the PDA study.
As discussed in Section 2.2, the design is well-balanced, as shown in Figure 202. Each attribute has an even distribution in the twelve card set. For instance, six cards represent large PDA’s and six cards represent small PDA’s. The same can be applied to the set of twelve cards for any of the six attributes.

2.3 Arriving at the Utility Function

The ultimate task of any conjoint study is to arrive at a utility function that describes the preference model of consumers. Naturally, that was one of the design goals for CARDS – to calculate the utility function based on a rank ordering of the 12 conjoint cards. Additionally, as stated earlier in this chapter, CARDS was designed to determine the utility function as quickly as possible. However, with 12! possible orderings and the internal inconsistencies associated with rank ordering product profiles, there was some room for simplification. By forcing respondents to be consistent (“consistent” by our standards), we were able to simply the process significantly. In section 2.3.1, the notion of consistency will be addressed.

2.3.1 Enforcing Consistency

Central to the design of CARDS is the idea of consistency. One of the problems commonly associated with rank ordering product profiles is that respondents tend to be inconsistent with respect to their preferences. Suppose in our PDA study that the user prefers a small PDA over a large PDA in one choice and does the opposite in another. CARDS eliminates the possibility of these inconsistencies. With our assumption of independence among attributes, such a preference is usually not possible. Moreover, CARDS assumes preferences are transitive. If \( i \) is preferred over
$j$ and $j$ is preferred over $k$, then it follows that $i$ is preferred over $k$. The system does not allow users to violate this property.

The assumption of linear utility functions has major implications in our system. It is possible for a respondent to order the 12 cards in a consistent, transitive, yet non-linear manner. CARDS deems all nonlinear utility functions inconsistent. This pose problems, particularly when certain attributes are closely correlated. Nevertheless, this is an approximation and a linear model, in most cases, is sufficient. By forcing a linear utility function, we further simplify the process, allowing for faster results.

Taking all these inconsistencies into account, CARDS eliminates the possibility of inconsistency by limiting the respondents' choice at each step to cards that do not violate our definition of consistency. Further details are presented in Chapter 3, the System Overview.

2.3.2 Discrete Approximation of Utilities

One of the major design considerations was the utility distribution. In CARDS, we approximate the multivariate utility distribution using discrete values. There were two versions to our study. In the first, each attribute (besides price) had 10 discrete levels associated with it. In the second, the granularity was 20. For each vector of utility values, we determined the card ordering that was consistent with those particular values. The orders and their associated coefficients were stored in a database. Respondents are forced to be consistent with the database. In other words, any order not contained in the database is deemed inconsistent.

The disadvantage to this approach is that the database contains only a small subset of the possible utilities. Forcing respondents to be consistent with the database could
compromise the accuracy of the utility function. Because CARDS was designed to arrive at a utility function as quickly as possible, these simplifying assumptions had to be made. However, despite its simplicity, CARDS provides accurate estimates of customers’ utility functions. Further details on how CARDS operates are provided in Chapter 3.
Chapter 3

System Overview

As Chapters 1 and 2 have pointed out, CARDS represents an adaptive approach to conjoint analysis that forces respondents to be consistent. In order to take on an adaptive approach and enforce consistency, a considerable amount of prior knowledge is required. In particular, all consistent utilities are calculated beforehand and stored in a database. During the questioning process, the user interface interfaces with the database at each step and determines the appropriate course of action. This chapter delves into more detail on the overall system, as well as the creation of the database.

3.1 A High Level View of CARDS

There are two main components to CARDS: the user interface and the database. Together, both components allow the system to display the properties of adaptivity and consistency. Figure 3-1 is an illustration of CARDS’ overall operation. The system operates with the following steps.
1. All twelve cards are shown on a web-based user-interface. The user chooses his/her first choice by clicking on the corresponding card.

2. CARDS queries the database of feasible orders.

3. The user interface shows only the cards that can possibly appear next, according to the orders in the database.

4. The user then chooses his/her favorite card from the ones remaining.

This process continues until a particular ordering in the database is matched. Both the user-interface and database are software-driven and are discussed in greater detail in the sections that follow.
3.2 User Interface

The growth of the internet, coupled with advances in computer technology have allowed for the possibility of web-based surveys. With the number of internet users growing daily, it now easier to reach customers today than ever before. Thus, web-based conjoint analysis is not just a luxury, it has developed into a virtual necessity. For this reason, CARDS has a web-based user interface.

The advantages of such web-based applications are rich, contextual, yet virtual media that can be used to illustrate products[2]. In our particular study, users could actually visualize the PDAs and better understand how they differ with various combinations of attributes. This allows respondents to get a better feel for what a particular product is like. Simple verbal descriptions of products may suffice in many cases; but adding a web-based interface takes users one step closer to the actual products. Another important advantage of a web-based interface is that the study is location-independent. Respondents can complete the study from the comfort of their own home, allowing product developers to reach a wider audience, thereby enhancing the quality of results.

To develop the user interface, PHP, a widely used scripting language, was used. The advantage of PHP is twofold.

- PHP can be directly embedded into HTML code.
- PHP allows for dynamic web pages.

The fact that PHP can be embedded into HTML is appealing because the adaptive features of CARDS can be placed within the presentation layer itself. That way, if there was a need to change some of the features of the system, the logic and the
look-and-feel can be changed simultaneously. The dynamic aspect of PHP allows
the system to adapt to the choices the user makes. In other words, the logic that
determines which cards to show on the screen is written in PHP. This is more efficient
than accounting for each and every possible scenario in the software.

3.3 Generating the Database

Figure 3-1 illustrates the importance of the database in CARDS. At each step, the
database is queried in order to determine the consistent cards that will appear on the
screen next. The database allows us to have prior knowledge of all feasible sort orders.
A database was generated for each version of the experiment: 10 levels per attribute
and 20 levels per attribute. The SQL database stores the following information:

- All feasible card orders.
- The average utility for each attribute (excluding price) for each card order.
- The sum of squares (variance) for each attribute for each card order.
- The number of utilities that generate each particular order.

Generating the database required a software solution. This section covers the software
issues involved in calculating utilities, storing pertinent information, and maintaining
consistency.

3.3.1 Data Structures

As mentioned in Section 2.3.2, a database was created for 10 and 20 levels. The
attributes and their associated levels are inputs to the software as a multidimensional
A multidimensional array offered the necessary flexibility for CARDS. First, from a development standpoint, it was the most visually intuitive solution. Attributes and their levels can be treated essentially as a table. Furthermore, because all the utility data is placed within the same data structure, managing the input data is much easier. The alternative was storing a separate data structure for each attribute. However, in terms of scalability, increasing the number of attributes may incur extra overhead. Table 3.1 is a visual representation of the multidimensional array used.

<table>
<thead>
<tr>
<th>Large Size</th>
<th>Cell</th>
<th>Hands Free</th>
<th>Battery</th>
<th>Wireless</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$636.565</td>
<td>$1981.647</td>
<td>$795.896</td>
<td>$761.028</td>
<td>$1481.216</td>
</tr>
</tbody>
</table>

Table 3.1: An $k \times n$ array, where $k$ is the number of levels and $n$ is the number of attributes. In CARDS, $k$ took on the value of either 10 or 20.

The cards are encoded in a similar fashion. Because the attributes in the PDA study are binary, the cards consist of 0 or 1 values for each attribute (1 denoting that the attribute is a “yes” and 0 a “no”). For instance, as shown in Table 3.2, Card 3 represents a PDA that is large, has cell phone and hands-free functionality, is wireless but has no extended battery.

### 3.3.2 Performing the Calculations

In our approach each and every possible vector of utilities is used to calculate a consistent ordering of the cards. Thus, the software was calculation-intensive. Below is the pseudo-code that illustrates the iterative process by which calculations are performed.
Table 3.2: An $c \times n$ array, where $c$ is the number of cards and $n$ is the number of attributes. This represents the encodings for the first five cards of the conjoint study that were introduced earlier in Chapter 2.

<table>
<thead>
<tr>
<th>Large Size</th>
<th>Cell</th>
<th>Hands Free</th>
<th>Battery</th>
<th>Wireless</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

CRUNCH_CALCULATIONS (n,k,c)
FOR each 5-tuple of Attributes DO
  FOR i = 1 to c DO
    Score Card[i]
  Sort Cards by Score
  Output Order to Buffer

Essentially, we iterate over the all possible 5-tuples of attribute levels and calculate a score for each card. Then the cards are sorted from high to low by score, generating a consistent card order. This order is then output to a buffer.

There are a total of $k^n$ 5-tuples. In the PDA study, we created a database where $k = 10$ and $k = 20$. With five attributes, the total number of orderings are $10^5$ (100000) and $20^5$ (3.2 million) respectively. It is important to note that these numbers do not imply that the orderings are unique. In fact, as we will discuss later, a large fraction of the orderings have multiple sets of utilities that generate them.
3.3.3 Scoring the Cards

Calculating the score for each card is essentially a dot-product operation. In the previous section, we established that for each possible 5-tuple of attributes, all cards are scored and then sorted from high to low by score. The formula can be expressed as follows:

\[
Score[i] = \left[ \begin{array}{c} att_1 \\
att_2 \\
\vdots \\
att_n \\
\end{array} \right] \cdot \left[ \begin{array}{c} card[i]_1 \\
 card[i]_2 \\
\vdots \\
 card[i]_n \\
\end{array} \right] - Price[i]
\]

In other words, the score for card \( i \) is calculated by taking the dot product of the 5-tuple vector and the card vector and subtracting off the price associated with card \( i \). The price is subtracted from the overall utility because intuitively, people are more receptive to lower prices. Thus, the higher the price, the greater the amount subtracted from the overall utility of that card.

As an illustrative example, suppose that we use the 5-tuple from Table 3.1 and we want to calculate the score or Card 2.

\[
Score[2] = \left[ \begin{array}{c} -$45.5 \\
 -$77.27 \\
 -$12.7 \\
 -$54.2 \\
 -$66.9 \\
\end{array} \right] \cdot \left[ \begin{array}{c} 0 \\
 1 \\
 0 \\
 1 \\
 1 \\
\end{array} \right] - $249
\]

\[
Score[2] = -$77.27 + -$54.2 + -$66.9 - $249 = -$449.27
\]
Once the scoring process is complete, we have an array of card scores, where the index of the array corresponds to the card number. The array is then sorted and is output to a buffer.

### 3.3.4 Building and Managing the Buffer

For each 5-tuple, we arrive at an array with the cards in order by score, from high to low. The information is then output and saved to an external buffer for later use. On each line, the buffer stores the following information:

- The twelve cards in order by score.
- The attributes (or 5-tuple) that generated that ordering.

Since the software iterates through all \(k^n\) possible combinations of attributes, there are a total of \(k^n\) lines in the buffer. When complete, the buffer contains the information from all possible 5-tuples.

<table>
<thead>
<tr>
<th>Line 1</th>
<th>(C_1)</th>
<th>(C_2)</th>
<th>(\cdots)</th>
<th>(C_{12})</th>
<th>(Att_1)</th>
<th>(Att_2)</th>
<th>(Att_3)</th>
<th>(Att_4)</th>
<th>(Att_5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 2</td>
<td>(C_1)</td>
<td>(C_2)</td>
<td>(\cdots)</td>
<td>(C_{12})</td>
<td>(Att_1)</td>
<td>(Att_2)</td>
<td>(Att_3)</td>
<td>(Att_4)</td>
<td>(Att_5)</td>
</tr>
<tr>
<td>\vdots</td>
<td>(C_1)</td>
<td>(C_2)</td>
<td>(\cdots)</td>
<td>(C_{12})</td>
<td>(Att_1)</td>
<td>(Att_2)</td>
<td>(Att_3)</td>
<td>(Att_4)</td>
<td>(Att_5)</td>
</tr>
<tr>
<td>Line (k^n)</td>
<td>(C_1)</td>
<td>(C_2)</td>
<td>(\cdots)</td>
<td>(C_{12})</td>
<td>(Att_1)</td>
<td>(Att_2)</td>
<td>(Att_3)</td>
<td>(Att_4)</td>
<td>(Att_5)</td>
</tr>
</tbody>
</table>

Table 3.3: The buffer to which all card orders and their corresponding attributes are written. Each line represents an ordering resulting from one of the \(k^n\) possible 5-tuples.

This buffer, however, is only temporary. One problem in using this as our database is that the \(k^n\) orderings are not all distinct. In fact, one of the fundamental motivations behind CARDS is the occurrence of repeats. How to handle repeats?
Theorem 1 Convex combinations of utility functions that yield the same ordering will also yield the same order.

In other words, suppose 12 utilities yield the same order. If we were to average their part-worths, the new utilities would also result in the same orderings of the cards. Using this intuition, we were able to create a database containing only the “unique” card orders. To create the database, we went through the following steps.

1. Sort the entire buffer by card order so that all repeat card orders are grouped together.

2. Scan the sorted buffer, line by line, combining all repeats into a single line.

3. Keep track of how many utilities generate each ordering.

4. For each ordering, calculate the mean of each attribute and the sum of squares.

5. Output each ordering and associated means, sum-of-squares and number of utilities to a table, based on the start card.

A typical entry of the database would look as follows:

| $C_1$ | ... | $C_{12}$ | Mean($Att_1$) | Sumsq($Att_1$) | ... | Mean($Att_5$) | Sumsq($Att_5$) | numutils |

Table 3.4: The buffer to which all card orders and their corresponding attributes are written. Each line represents an ordering resulting from one of the $k^n$ possible 5-tuples.
Each ordering is processed and inserted into a particular table of the database. In CARDS, there are 12 tables in all and each table had orders with the same start card.

Figure 3-2: Once each *unique* order in the buffer is processed, the ordering (along with accompanying statistics) are written to the database.

### 3.3.5 Sorting

There were two types of sorts employed in the software. One for the cards to arrange them in order, from high to low. For this particular sort, we used Bubble-Sort. Bubble-sort compares each element with its nearest neighbors and switches their location if they are out of order. With a running time of \( O(n^2) \), this is by no means an optimal sorting algorithm. However, with 12 cards, the difference between good and bad sorting is merely a few clock-cycles. In the event that there are a lot of cards a better choice might be "merge-sort."

We are not afforded such luxury when managing the temporary buffer. With \( k^n \) lines of data to sort, a comparison sort that makes \( n^2 \) comparisons is too slow. Thus, we employ quick-sort. Quick sort runs in \( O(n \log n) \) time in the average case. The fact that it is faster than the conventional comparison sort and is built into the C++ library made this an easy choice.
3.3.6 Observations

When the database was completed, we had a few notable observations.

- **Few Unique Orderings.** An overwhelming majority of the orderings generated from the attribute combinations appeared more than once. Thus, only a small percentage of the orders were unique. In the $10^5$ database, only 34,563 (35 percent) were unique. In the $20^5$ database, the number was 168,532 (5 percent).

- **Card 2 is the Favorite.** An overwhelming majority of the orderings fell in the table with Card 2 as the top choice. This makes sense, considering that Card 2 had all the features for a cheaper price. The closest runner-up to Card 2 was Card 4, whose table had half the number of records.

- **Average Utilities.** We analyzed some of the utilities that appeared many times, and looked at the mean values for each attribute. We fed those utilities into CARDS and in each case, it yielded the same order, thereby substantiating the theorem in section 3.4.4.

3.4 Software Issues

In addition to the sorting, there are a number of software issues and constraints that are pertinent. Considering that the number of iterations is exponential in the attribute granularity ($k^n$), and at each iteration there is a considerable amount of File I/O being done, the software is calculation-intensive.

A significant amount of buffer space is needed in order to perform all the calculations necessary for making the database. At one point, the temporary buffer and all the
tables (the number of tables equals the number of cards) are stored in memory because they interface with each other for many of the mean and variance calculations.

The software is designed to scale to a variable number of attributes and levels seamlessly. The number of attributes and granularity are declared as constants in the declaration section of the software. If the number of levels were to change, only the constant needs to be changed. However, if the number of attributes change, the depth of the iteration needs to change as well. Nevertheless, when it comes to modifying the actual code to accommodate different studies, the software scales with relative ease.

However scaling with respect to speed is a major concern. Simply increasing the number of attributes by one will incur a significant cost on speed. Suppose that the total number of possible 5-tuples were \( k^n \). Increasing the number of attributes to \( n+1 \) should increase the number of possible 5-tuples by a factor of \( k \). In other words in the PDA study, the number of combinations increases from 3.2 million to 60.8 million. Because the software iterates over all possible combinations of attributes, the number of iterations increases by a constant factor, \( k \) (an additional 57.6 million iterations!). 57.6 additional iterations means 57.6 more lines in the intermediate buffer file, which in turn adds a more overhead to the quick-sort procedure which sorts the entire buffer. Moreover, this calls for many more mean and variance calculations. As we can see here, simply increasing the number of attributes by one has somewhat of a domino effect.
Chapter 4

Conclusion & Future Work

This project involved the development of a new adaptive approach to conjoint analysis in hopes of arriving at estimates of customers’ utility functions both quickly and accurately. The delineating feature of our system is the notion of consistency. CARDS not only forces respondents to be consistent with respect to order, nonlinear utilities are considered inconsistent as well. Moreover, a discrete approximation of the utility distribution allows us to generate a database of consistent utility functions quickly.

The nature of a discrete approximation however, allows for the possibility of a “miss.” Some utilities, despite being linear and consistent with respect to order, are deemed inconsistent in CARDS simply because they don’t exist in the database. In a sample of 240 respondents, it was found that 35 percent of their responses (which were linear and consistent in terms of order) did not appear in the database. This is a byproduct of the simplifying assumptions that have been made in order to maintain consistency. Despite this, the CARDS estimate of the utility function is not far off, displaying a mean correlation of 0.89 – a high metric considering the degree to which the process has been simplified.
The essence of CARDS is its simplicity. At the very beginning of the task, respondents are shown all 12 possible conjoint cards. As they begin the process of selecting the cards in order of preference, the system eliminates the cards that are inconsistent. Thus, in the great majority of cases, makes fewer selections and the system arrives at an answer faster. The number of clicks is a solid metric for a web-based conjoint study. With a value of 8, CARDS reduces the number of clicks by 4 on the average case. In many cases, the reduction is even more prominent.

Nevertheless, this can be a double-edged sword. Our system places a higher weight on earlier cards. For instance, suppose the respondent calculates Card 3 as her first choice. Immediately, the system eliminates all orderings except for those beginning with 3. In essence, ordering the cards is like traversing a tree. Once the fist card is selected, the respondent is sent down a particular path. If the first card was not accurately selected, the system produces a poor estimate of the utility function. The assumption was made, however, that respondents are typically more confident about their first choice rather than their middle choices.

There is in essence a proof of concept and there is plenty of room for future work in the area. One area to explore is random sampling. Instead of calculating the orderings for each and every vector as we did in the discrete approximation of the utility distribution, we can maintain a database of orderings generated from random samples of utility vectors.

In Chapter 1, utility functions were defined in terms of a linear program. An area to explore with CARDS is an LP-based approach in which consistency is tested dynamically, rather than a priori. In our current approach, consistent utilities are stored in a database and provide the system with prior knowledge. In an LP-based approach, the need for a database is eliminated altogether. Instead, as each card is selected, the
LP, using constraints generated from previous selections, will dynamically test every possible next card to see if they are consistent. The approach can be summarized as follows.

1. Respondent chooses card at each step; Current order is $i_1, i_2, \ldots, i_k$.

2. Suppose the set $S$ denotes the remaining cards. Solve $S$ LPs for each card in $S$ to see which cards are consistent and can be next.

By sampling randomly over the LP feasible region, such useful statistics as the mean and standard deviation can be determined.
Appendix A

C++ Code

All software will be included here.

/* The subroutine “calc” is supposed to score all the cards and output to to the file. */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* Defines */

/* Number of cards in the simulation. */
#define NUM_CARDS 12

/* Number of attributes a given card can take. */
/* The attributes are: Large Size, Cell, Hands free, Battery, Wireless */
#define NUM_ATTRIBUTES 5

/* The number of possible values a given attribute can take. */
```c
#include ATTRIBUTE_.GRANULARITY 20

/* The number of floats to include in an internal data line */
#define BLOCK_LINE (NUM_CARDS + NUM_ATTRIBUTES)

/* Globals */

/* These are the attributes and their associated levels. Each column represents a distinct attri

float attributes[ATTRIBUTE_.GRANULARITY][NUM_ATTRIBUTES] =
{
  {-586.331, -182.972, -193.869, -226.960, -355.634},
  {-501.292, -32.815, -125.210, -80.210, -82.799},
  {-430.672, 91.882, -68.193, -28.137, 14.015},
  {-368.534, 201.601, -18.024, 17.682, 99.200},
  {-311.779, 301.817, 27.799, 59.532, 177.007},
  {-258.521, 395.856, 70.798, 98.802, 250.018},
  {-207.472, 485.996, 112.015, 136.445, 320.002},
  {-157.648, 573.972, 152.241, 173.184, 388.307},
  {-108.218, 661.253, 192.150, 209.632, 456.071},
  {-58.394, 749.230, 232.377, 246.371, 524.375},
  {-7.345, 839.370, 273.593, 284.014, 594.359},
  {45.913, 933.409, 316.593, 323.284, 667.371},
  {102.668, 1033.624, 362.416, 365.134, 745.178},
  {164.806, 1143.344, 412.585, 410.953, 830.363},
};
```
{235.426, 1268.040, 469.602, 463.026, 927.177},
{320.465, 1418.198, 538.261, 525.732, 1043.758},
{434.443, 1619.453, 630.284, 609.776, 1200.011},
{639.565, 1981.647, 795.896, 761.028, 1481.216}
};

int cards[NUM_CARDS][NUM_ATTRIBUTES] =
{
  {1, 0, 0, 0, 0},
  {0, 1, 0, 1, 1},
  {1, 1, 1, 0, 1},
  {0, 0, 1, 1, 1},
  {1, 1, 0, 1, 1},
  {1, 0, 1, 0, 1},
  {1, 0, 1, 0, 1},
  {1, 0, 0, 1, 0},
  {0, 0, 0, 1, 0},
  {0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0},
  {0, 0, 1, 0, 0},
  {0, 1, 1, 0, 0},
  {1, 1, 1, 1, 0},
  {0, 0, 1, 1, 1}
};

/* The prices of Cards 1-12 */
int price[NUM_CARDS] = {499, 249, 249, 249, 499, 249, 249, 499, 249, 499, 499, 499, 499};
FILE *writefile[NUM_CARDS];

char outfile_path[NUM_CARDS][50] =
{
    "one.txt",
    "two.txt",
    "three.txt",
    "four.txt",
    "five.txt",
    "six.txt",
    "seven.txt",
    "eight.txt",
    "nine.txt",
    "ten.txt",
    "eleven.txt",
    "twelve.txt"
};

/* Function Prototypes */

/* Normally a sub-par sorting algorithm...only using because, with twelve items tops, the di */
void bubble_sort( int *scores, int *order );

/* Calculate pertinent values and insert into all */
void calc( float *all, float *these_attr, int all_pos );
Function to compare one line of data against another */

```c
int compare_orders( const void *first_v, const void *second_v );
```

Takes in the humongous block of lines
<br>
<\[num-cards\] card order> <\[num-attributes\] attribute values>
and outputs to \[num-cards\] separate files
<br>
<\[num-cards\] card order> <means and squares of attrs> <\# times occurring>

`*/

void write_block_to_files( float *huge_block, int *huge_block_order );`

Globals */

```c
int *all_order;
float *all_data;
```

Main */

```c
int main()
{
    float curr_attribs[NUM_ATTRIBUTES];
    int x[NUM_ATTRIBUTES], y;
    int c_attr;
    int all_data_pos = 0;
    long num_lines = pow(ATTRIBUTE_GRANULARITY, NUM_ATTRIBUTES);

    /* Initialize our huge arrays */
```
all_data = new float[ BLOCK_LINE * num_lines ];

if ( all_data == NULL )
{
    /* Could not allocate this huge buffer...how shocking! */
    fprintf( stderr, "Could not allocate buffer of size %d.\n", BLOCK_LINE * num_lines );
}

all_order = new int[ num_lines ];

if ( all_order == NULL )
{
    /* Could not allocate this huge buffer...how shocking! */
    fprintf( stderr, "Could not allocate buffer of size %d.\n", num_lines );
}

for ( y = 0; y < num_lines; ++y )
    all_order[y] = y;

/* Actually crunch out the calculations. */

for ( x[0] = 0; x[0] < ATTRIBUTE_GRANULARITY; ++(x[0]) )
{
    for ( x[1] = 0; x[1] < ATTRIBUTE_GRANULARITY; ++(x[1]) )
    {
        for ( x[2] = 0; x[2] < ATTRIBUTE_GRANULARITY; ++(x[2]) )
        {
            for ( x[3] = 0; x[3] < ATTRIBUTE_GRANULARITY; ++(x[3]) )
            {
                for ( x[4] = 0; x[4] <ATTRIBUTE_GRANULARITY; ++(x[4]) )
                {

for(c_attr=0; c_attr < NUM_ATTRIBUTES; ++c_attr )
{
    curr_atrribs[c_attr] = attributes[x[c_attr]][c_attr];
}
calc( all_data, curr_atrribs, all_data_pos++ );
}
}
}
}

/* Do sorting of data */
qsort( all_order, num_lines, sizeof(int), compare_orders );

/* Spit out actual results */
write_block_to_files( all_data, all_order );
delete []all_data;
delete []all_order;
return(0);
}

void bubble_sort( float *scores, int *order )
{
    int i;
    int j;
    int temp;
/ * Start the ranking completely unordered */
for( i = 0; i < NUM_CARDS; ++i )
    order[i] = i;
/* Sort them by looking up values in scores... */
for( i = 0; i < NUM_CARDS; ++i )
{
    /* Now finding (i+1)'th highest score. */
    for( j = (i + 1); j < NUM_CARDS; ++j )
        if ( scores[order[j]] > scores[order[i]] )
            {
            /* Switch 'em */
            temp = order[j];
            order[j] = order[i];
            order[i] = temp;
            }
}
/* Increment the order values by one, so the list goes from 1-12 instead from 0-11 */
for( i = 0; i < NUM_CARDS; ++i )
    order[i] = order[i] + 1;

void calc( float *all, float *these_attr, int all_pos )
{
    float score[NUM_CARDS];
    int card_order[NUM_CARDS];
}
```c
int current_card;
int current_attr;
int exact_allpos = all_pos * BLOCK_LINE;

/* Calculate scores for the cards under these conditions */
for ( current_card = 0; current_card < NUM_CARDS; ++current_card )
{
    score[current_card] = 0;
    for( current_attr = 0; current_attr < NUM_ATTRIBUTES; ++current_attr )
        score[current_card] += these_attr[current_attr]*cards[current_card][current_attr];
    score[current_card] -= price[current_card];
}

/* Sort the cards by score, high to low. */
bubble_sort(score, card_order);

/* Put data into all */
for( current_card = 0; current_card < NUM_CARDS; ++current_card )
    all[exact_allpos + current_card] = card_order[current_card];
for( current_attr = 0; current_attr < NUM_ATTRIBUTES; ++current_attr )
    all[exact_allpos + NUM_CARDS + current_attr] = these_attr[current_attr];
/* printf("Line %d is %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f \n", all_pos,
    all[exact_allpos+0],all[exact_allpos+1],all[exact_allpos+2],all[exact_allpos+3],
    all[exact_allpos+4],all[exact_allpos+5],all[exact_allpos+6],all[exact_allpos+7],
    all[exact_allpos+8],all[exact_allpos+9],all[exact_allpos+10],all[exact_allpos+11],
    all[exact_allpos+12],all[exact_allpos+13],all[exact_allpos+14],all[exact_allpos+15],
    all[exact_allpos+16] ); */
```
int compare_orders( const void *first_v, const void *second_v )
{
    int first = *((int *)first_v);
    int second = *((int *)second_v);
    int my_res = 0;
    int curr_card;
    int first_allpos = first * BLOCK_LINE;
    int second_allpos = second * BLOCK_LINE;

    for ( curr_card = 0; (curr_card < NUM_CARDS) && !my_res; ++curr_card )
    {
        if ( all_data[first_allpos + curr_card] < all_data[second_allpos + curr_card] )
            my_res = -1;
        else if ( all_data[first_allpos + curr_card] > all_data[second_allpos + curr_card] )
            my_res = 1;
    }
    return my_res;
}

void write_block_to_files( float *huge_block, int *huge_block_order )
{
    int c_attr;
    int c_card;
    int c_combo = 0;
    int chuge_block_pos;
    int c_ord_combo;
    int mean_multiplier;
}
float mean_attr[NUM_ATTRIBUTES];
long num_lines = pow(ATTRIBUTE_GRANULARITY, NUM_ATTRIBUTES);
int order_pos = 0;
float sumsq_attr[NUM_ATTRIBUTES];

/* Create/open files for writing. */
for( c_card = 0; c_card < NUM_CARDS; ++c_card )
{
    writefile[c_card] = fopen( outfile_path[c_card], "w" );
    if ( writefile == 0 )
    {
        fprintf( stderr, "Unable to write outfile %d in current dir. Check permissions? exit(-1);
    }
}
for( c_combo = 0; c_combo < num_lines; )
{
    c_ord_combo = huge_block_order[c_combo];
    /* Calculate means and squares */
    for ( c_attr = 0; c_attr < NUM_ATTRIBUTES; ++c_attr )
    {
        mean_attr[c_attr] = 0;
        sumsq_attr[c_attr] = 0;
    }
    mean_multiplier = 0;
    while ( ( c_combo < num_lines ) &&
( compare_orders( (void *)&c_ord_combo, (void *)&huge_block_order[c_combo];
{
    c_huge_block_pos = BLOCK_LINE * huge_block_order[c_combo];
    for ( c_attr = 0; c_attr < NUM_ATTRIBUTES; ++c_attr )
    {
        mean_attr[c_attr] += huge_block[c_huge_block_pos + NUM_CARDS + c_attr];
        /* sumsq_attr[c_attr] += mean_attr[c_attr] * mean_attr[c_attr]; */
        sumsq_attr[c_attr] += huge_block[c_huge_block_pos + NUM_CARDS + c_attr] * huge_block[c_huge_block_pos + NUM_CARDS + c_attr] * huge_block[c_huge_block_pos + NUM_CARDS + c_attr] * huge_block[c_huge_block_pos + NUM_CARDS + c_attr];
    }
    c_combo++;
    mean_multiplier++;
}
for ( c_attr = 0; c_attr < NUM_ATTRIBUTES; ++c_attr )
    mean_attr[c_attr] /= mean_multiplier;
/* Output to file */
fprintf( writefile[[(int)huge_block[c_huge_block_pos]]1], "%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d%
if ( fclose(writefile[c_card]) != 0 )
{
    fprintf(stderr, "File %d couldn't close!", c_card);
    exit(-2);
}
}
Bibliography


